2004-07

The Profession of IT, The Field of Programmers Myth

Denning, Peter J.

The Field of Programming Myth. (July 2004) The persistent public image of computing as a field of programmers has become a costly myth. Reversing it is possible but not easy.

Peter J. Denning

The Field of Programmers Myth

The persistent public image of computing as a field of programmers has become a costly myth. Reversing it is possible but not easy.

Are you concerned with the widely held public view of our field: "Computer science equals programming"? You ought to be. It is firmly lodged in movies, novels, news reports, advertisements, political speeches, perceptions of other scientists and engineers, and in the minds of prospective entrants to our profession. What's more, many people believe computer science is only a technology field without much science and engineering of its own; the word "science" in our title is undeserved. Want to know why this is so and how you can help? Read on.

The story that computer science equals programming is old. By the mid-1980s it had become sufficiently irksome that the ACM/IEEE-CS committee on the core of computing made it an objective to dispel this notion. Their report, *Computing as a Discipline*, depicted the field with the help of a 9x3 matrix, showing nine core areas and three processes (theory, abstraction, and design) [2]. They put programming as a subpart of the design component of most of the nine areas, especially algorithms and software engineering. Their model became the backbone of the ACM Curriculum '91 recommendation. But the old story endures.

In the past few years, the story has become a major impediment. Many computing people are no longer irked; they are infuriated. Many programming and software engineering jobs were lost in the downsizing following the Internet business crash in 2001. But many jobs did not come back in the recovery; they were outsourced as companies took advantage of significantly lower labor rates in other countries. Enrollments in computer science and engineering dropped 30% between 2002 and 2004. The dropout rates of those who enroll hover between 35% and 50%. Women continue to prefer other fields.

Prospective students ask, "If the heart and soul of computing (programming) is being auctioned off to the lowest offshore bidder, what is the future for me?" What do we say to them? In an effort to persuade them to choose computing majors, Microsoft's Bill Gates made an extraordinary tour of several U.S. universities earlier this year to tell

CYRIL CABRY

Our challenge is to adopt a larger view of the field that reveals the science and does not confuse science and practice.

students about the numerous computing jobs that will never migrate offshore.

Where does the programming story come from? Why does it persist? The short answer is: we are reaping what we have sown. It is rooted in the logic that computers need programs to work and therefore that programming is fundamental. This logic pervades our curricula and how people learn computer science. Our first courses are programming. The many course projects are often called "programming projects"—not design projects, database projects, network projects, or graphics projects. When forming opinions about their major field in college, high school students look at their high school courses in computing and see…programming. Those interested in advanced placement see…object-oriented programming with Java.

Media outlets tell many stories about computing. Who created and distributed modern public key cryptography? Programmers. Who instructed the dozens of microprocessors in your car? Who created the software that analyzes your MRI scan? Who created the Web browser? Who wrote the SETI code that helps search for extraterrestrial intelligence when your workstation is idle? Programmers all. Who breaks into systems? Hackers, a sect of rogue programmers. Who writes viruses and worms? Who launches denial-of-service attacks? Defaces Web pages? Hijacks computers and turns them over to spammers? More rogue programmers. Who made an error that caused a Mars probe to crash? A programmer. It's everywhere, gang. The good and bad, all done by programmers.

Most people are not seeing the stories of the computer architects, the network engineers, the operating systems engineers, the database engineers, the graphics specialists, the software architects, the software system designers, the security experts, the simulators, the virtual realitors, the supercomputing experts, the roboticians, and many more. All the stories are being told as if done by programmers. Bill Gates is alarmed by this. We should be too.

How might we reverse this? I offered a solution in my Nov. 2003 column, "Great Principles of Computing" [4]. The great principles framework is a new portrayal of our field that emphasizes our scientific and engineering principles and our four core practices. Our fundamental principles are in design and in the mechanics of computation, communication, coordination, recollection, and automation. These principles were not borrowed from other fields; computer scientists developed them. Our four core practices are programming, engineering of systems, modeling, and innovating. Looking into this world through a programming window is like viewing a construction site through a peephole in a fence: you can't see much.

Our challenge is to adopt a larger view of the field that reveals the science and does not confuse science and practice.

Programming Has Not Been Industry's Central Problem for a Long Time

When the field was young (during the 1950s) and much smaller, the primary concern was build-

ing computing systems. The two most common jobs were computer architect and programmer. Many industry leaders saw how labor intensive programming was and asked universities to teach it from the start so their graduates could fill programming jobs. The primary intellectual focus was on the logic of hardware and software, and the primary professional practice was programming. Programming was seen as the only available means to translate computing concepts into functioning reality.

In 1968, NATO sponsored a historic meeting of academic and industry leaders to discuss what they dubbed the software crisis. They saw software systems rapidly growing in size and complexity and being put into applications where failures could cost lives and ruin businesses. They believed that the fundamental notion behind programming—that programs implement mathematical functions—could not cope with the complexity and fuzziness of requirements in real, large, safety-critical applications. They believed the future challenges of software development would be to devise engineering processes to translate complex requirements into working systems, to deal with fuzzy and shifting requirements, to assess and manage risk, to systematize the process of locating and eradicating errors, to organize and manage teams of programmers, and to satisfy customers. They called for the formation of a new discipline: software engineering.

One of the unspoken conclusions of that meeting was that whatever we were teaching under the heading of programming was inadequate and that a fundamentally new, systems-oriented engineering approach would be needed.

Seventeen years later, Fred Brooks took stock of the progress of software engineering in his landmark critique, "No Silver Bullet" [1]. How far had we advanced toward the goal of systematically building reliable, dependable, and useful computing systems? He maintained that the fundamental obstacle to this goal was, and always will be, the complex behavior of large software systems. Much of the work in software engineering—in languages, tools, graphics aids, debugging aids, structured programming, coding automation, and the like—had been impressive. Nonetheless, he said, these technologies proved only marginally helpful against the core problem of software development: getting an intellectual grasp on the complexity of the application. He concluded that software production is inherently a talent-and-design problem that can only be met by a determination to identify and cultivate great designers.

Brooks did not say "cultivate great programmers"; his concern was the design of systems. We now understand that individual talent and skill have a huge effect on the quality of a software developer's work: good developers are often 10 times as productive as novices, and a few virtuosi may be 50 times as productive. Is great productivity a gift granted to a select few, or a skill that can be learned? Brooks called on the community to mount a concerted effort to teach computing people to be great designers and expert software developers. Despite the wide praise and admiration for this sentiment, few have responded to his challenge.

In 1989, Edsger Dijkstra, one of the giants of our field and a passionate believer in the mathematical view of programs and programming, debated with his peers on the right way to teach computing science [6]. He maintained that the art of programming is the linking thread that gathers disparate branches into a single discipline. Over the previous quarter-century, he had formulated many of the great intellectual challenges of the field as programming—the goto statement, structured programming, concurrent processes, semaphores, deadlocks, recursive programming in Algol, and deriving correct programs. He advocated that we remove all real-world programming languages from the beginning courses and teach instead the formal derivation of

programs from logical predicates that express their requirements. His critics thought this approach was too limiting and advocated an approach closer to software engineering. Most of our curricula today do neither: programming is not taught as construction of algorithms for mathematical functions or as a step toward software systems engineering; it is taught as an introduction to an industrial-strength language, Java or C++.

Numerous studies have documented the dismal success statistics of software projects: approximately one-third are delivered on time and within budget; another one-third are delivered late or over budget; and the remainder are not delivered at all. The authors of the studies invariably conclude that the errant projects go astray because of various people problems—dysfunctional teams, breakdown of interpersonal relationships, inadequate listening to the customer. Very few software engineering courses take up these issues and teach students good requirements, error detection, team, and customer practice. Why? Many faculty members see these aspects as "soft" and believe there isn't even enough room in the curriculum for all the essential "hard" core technical material.

In *The Unfinished Revolution*, Michael Dertouzos documented 15 common, persistent design flaws in computing systems [5]. We have known about them for years but our approaches to teaching software development have been ineffectual—we seem to have a blindness that is passed on to each new generation of students. He recommends moving from product-centered software development to human-centered development.

Because we think of programming as a process that yields programs, we grade programs; in the future we will also grade programmers.

So we still have a long way to go. We are captured by a historic tradition that sees programs as mathematical functions and programming as the central practice for translating these functions into working systems. We have only partially embraced the newer tradition that sees programs as components of complex systems that must be designed under severe constraints. Industry has pushed us for a long time toward the newer tradition but we haven't let go enough of the older one to get there.

Computing Practices

Over the years, the practices of systems, modeling, and innovating have taken on peer roles with programming [4]. Today's computing professional must be competent in all four practices, but with different relative emphasis depending on individual specialty and style. For example, HCI people place a great deal of emphasis on design, modeling, and simulation, and hire programmers to do the coding. Computer architects use CAD systems but do little programming. Many software engineers spend their time interacting with customers and designing systems, but not coding. There are numerous other examples. Today, programming is neither the dominant practice nor the defining practice.

I have four recommendations to help reorganize the curriculum for the teaching of all four practices in a coherent way that does not confuse them with principles.

1. Teach algorithmic thinking. Algorithmic thinking is a mental practice of engineering and scientific discovery that conceptualizes problems with digital representations and seeks algorithms that express or find solutions. Algorithmic thinking pervades and supports all four core practices. Why not teach this in our first course(s) and teach programming separately (next recommendation)? Lace the course(s) with many powerful stories of great inventions and innovations.

2. Group the teaching of practices into a Computing Practices section of the curriculum. The courses now labeled Computer Science 1 and 2, CS1 and CS2, could in most cases be relabeled Programming Practices, PP1 and PP2. Introduction to software engineering can be modified to be introduction to systems practices. A modeling practices course will need to be added to most curricula. The increasingly popular capstone design course can also teach innovation practices.

3. Structure a teaching framework around a ladder of competence in the practice. In any domain, the ladder of competence describes the recognized levels of skill that one can attain given enough practice and experience. They are: beginner, advanced beginner, competent, proficient, virtuoso, master, legend [3]. Define criteria that would permit us to judge the level at which someone is performing. Grade both the quality of the work and the quality of the performance. Get teachers with industry experience for these courses, and encourage faculty members to take industry sabbaticals and leaves of absence.

Applying this to programming, we can see different criteria at each level. The beginner would focus on learning the syntax and execution rules of programs and the basic methods of scaffolding to help detect errors. The competent programmer would have extensive knowledge of libraries and basic algorithms and would be able to bring many modules together into a system that satisfies customers. The proficient programmer would be facile in many programming languages, would see individual languages as ways to express algorithms that are already pictured in the mind, and would by example set standards of excellence in programming that others would admire and follow. The master would have extensive knowledge of the historical developments in programming, would be able to design large software systems combining many levels of abstraction, and would define new methods that improve all programming practice. Professional societies such as ACM and IEEE-CS would issue guidelines for certification at the various levels. Today, because we think of programming as a process that yields programs, we grade programs; in the future we will also grade programmers.

From this perspective it is important to ask what is an appropriate first language for a beginner. Although competence in Java might be something expected of BS-degree graduate, is Java suitable for beginners? Why not start with simpler steps that enable students to learn the practice first as beginner, and experience the joy of success? Mark Guzdial's course on Media Computation at Georgia Tech is an example [7]. It uses Python, a much simpler object-oriented language. While it may be too late to reverse the process of pushing Java into the Advanced Placement curriculum in high schools, it is not too late to influence the AP curriculum to give greater emphasis to algorithmic thinking and problem-solving—an appropriate starting point for beginners.

4. Teach the practice of error detection and correction. Even the most skilled people make mistakes. Making mistakes is part of the process; detecting, correcting, and learning from them is part of the practice. Reminiscing about his early days on EDSAC in the 1940s, Maurice Wilkes wrote in 1979: "As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to

be spent in finding mistakes in my own programs."

From Abstractions to Actions

It may be asked: why teach programming early? Why not teach the science and algorithmic thinking first, programming later? I do not recommend this approach.

Most computing professionals do not appreciate how abstract our field appears to others. We have become so good at defining and manipulating abstractions that we hardly notice how skillfully we create abstract "objects," which upon deployment in a computer perform useful actions. Students need to see from the beginning how to connect abstractions to actions. There is little joy in worlds of pure abstractions devoid of action. The practices of programming and systems make this connection.

A Great Principles Library

To support computing people and outsiders in learning a principles-oriented portrayal of computing, I propose the establishment of a Great Principles Library. The Library would contain sections for the great principles (design and mechanics), the core practices (programming, systems, modeling, and innovating), and the core technologies. Each section would offer tutorial materials for beginners, intermediate, and advanced practitioners; seminal papers; historical summaries of the evolution of principles and practices; stories of great inventions and innovations. It would be overseen by a board of editors who would commission new items and the cross-referencing of existing published items, and who would see to the quality, consistency, and accuracy of the materials included. It would be part of the ACM Digital Library.

These steps—revamping the teaching of programming and creating a repository of great-principles materials—would go a long way to dispel the myth that computer science equals programming. C

References

1. Brooks, F.P., Jr. No silver bullet. In *The Mythical Man Month*, Chapters 16 and 17, Addison-Wesley (1995 edition).
2. Denning, P.J. et al. Computing as a discipline. *Commun. ACM* 32, 1 (Jan. 1989), 9–33.
3. Denning, P.J. Career redux. *Commun. ACM 45,* 9 (Sept. 2002), 21–26.
4. Denning, P.J. Great principles of computing. *Commun. ACM 46,* 11 (Nov. 2003), 15–20.
5. Dertouzos, M. *The Unfinished Revolution*. Harper Collins, 2001.
6. Dijkstra, E. et al. A debate on teaching computing science. *Commun. ACM 32*, 12 (Dec. 1989), 1397–1414.
7. Guzdial, M. and Solloway, E. Computer science is more important than calculus: The challenge of living up to our potential. *Inroads* (ACM SIGCSE Bulletin), June 2003, 5–8.

Peter J. Denning (pjd@nps.edu) is the director of the Cebrowski Institute for information innovation and superiority at the Naval Postgraduate School in Monterey, CA, and is a past president of ACM.